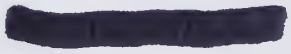


ROBOCAR: EDUCATING THE LAYMAN IN COMPUTER SCIENCE

by

James W. Renshaw

July, 1975

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-75-741

ROBOCAR: EDUCATING THE LAYMAN IN COMPUTER SCIENCE

by

James W. Renshaw

July, 1975

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

This work was supported in part by the National Science Foundation under Grant No. US-NSF-EC-41511 and was submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

ACKNOWLEDGMENT

I would like to thank Dr. Jurg Nievergelt for his thoughtful guidance and support throughout this project. Without his many suggestions it would never have gotten off the ground.

Thanks also to Sandy's Typing Service and to my wife for their careful proofreading and typing of this thesis.

And finally, my appreciation to the Department of Computer Science which supported my studies and to the National Science Foundation under whose grant EC-41511 this project was supported is sincere and considerable.

TABLE OF CONTENTS

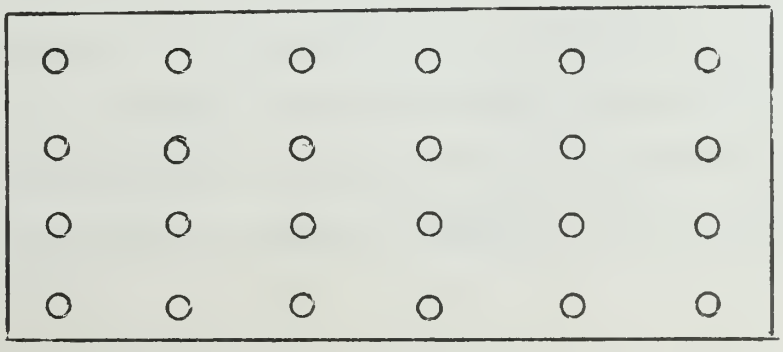
INTRODUCTION

This thesis describes a sequence of PLATO lessons designed to teach computer programming. The following two ideas have guided the design of this instructional material: First, to create an environment where many of the fundamental concepts of programming can be illustrated in a simple, intuitive setting, and second, to use the facilities provided by the PLATO IV computer-based education system to their fullest extent for presenting information and allowing the student to practice immediately what he has learned, by means of a dialog which consists primarily of animated graphics.

The setting chosen to illustrate the fundamentals of programming is that of a robot cab which, when given the proper sequence of instructions, may be made to move along the streets of a simulated city.

The robot cab looks like this: 

And the city is represented in a form similar to the one below, although it may vary slightly in different presentations.



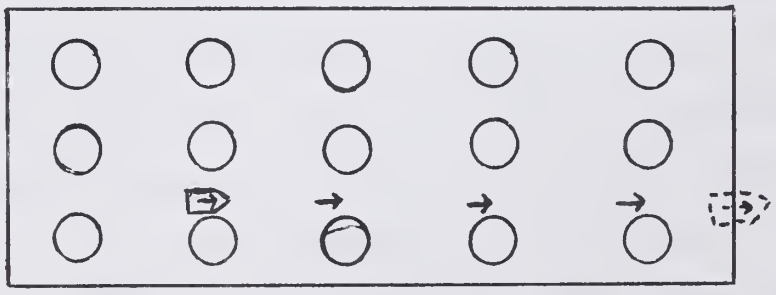
The robot cab simulates a computer, with its movement around the city the discernable output or action associated with a computer. The instructions the cab can carry out are called the ROBOCAR language. They are explained in detail in Appendix A of this document.

The following program will, when executed, cause the cab to go forward through all the intersections, stopping only when it has penetrated the city limits and is physically not allowed to proceed further.

The program is :

```
a  fwd
   goto  a
```

consisting of the two statements above. Given the initial placement of the cab in the city below,



its successive new positions once the program is run are marked by the arrows. The cab stops where its dotted outline is visible. As mentioned before, the program would terminate in error because the cab penetrated the city limits, a condition not allowed in our simulated city.

With this brief introduction as a preview, you will find now the discussion of the goals, justification and implementation of the ROBOCAR sequence of lessons.

1. GOALS AND JUSTIFICATION

The ROBOCAR sequence of lessons is a part of the Automated Computer Science Education System (ACSES), Nievergelt (1975). The intent of this series of four lessons and supporting programs is to educate anyone with an interest in computer science in some fundamental principles and techniques of computer programming and problem solving.

As far as background is concerned, one hopefully needs only the interest mentioned above, the ability to operate a PLATO terminal and to read the material presented in the lessons. It is hoped, then, that business and professional people, university students as well as higher elementary and secondary pupils will be able to benefit from this offering. For example, during the Fall 1974 term at the University of Illinois, students in the course CS 106 (Introduction to Computers for Teachers) have used the ROBOCAR sequence.

2. BENEFITS OF A MINI-LANGUAGE

The vehicle to carry out the above purposes is a language called, not surprisingly, the ROBOCAR language. Designed primarily by Dr. Jurg Nievergelt of the University of Illinois, with suggestions and modifications by myself and others, this programming language is one among a number of "mini-languages" available in the ACSES instructional system.

Using a mini-language for presenting a layman's introduction to programming can be justified by at least two of the qualities which the ROBOCAR language possesses: small size and simplicity. These characteristics, although not completely independent, serve to allow a steady, if not swift pace for the student's growth in his programming achievements.

The smallness and simplicity of the language have been achieved mainly by fashioning an appropriate environment. An environment was chosen which allowed a small instruction set to exist, but which lent itself to a variety of interesting and reasonably challenging problem-solving tasks. By limiting the scope and complexity of the simulation and making the robot cab simple-minded, the environment and thus the instruction set was kept small.

Another simplifying measure was the decision to omit variables from the language in order to avoid any discussion

of such concepts as declaration and type of variables. The implementation of nested loops with counters, which occur frequently in typical ROBOCAR programs can be handled well by the use of one stack. Knowledge of the workings of a stack will be useful in further study of computer science.

Designing the language so that all instructions are indivisible (no instruction subsumes the function of any other instruction) is another way that the number of instructions was kept small. With a small-sized language, the student need only consider a few instructions to perform a task that faces him. Narrowing the choice of instructions should lead to faster solving of programming problems.

Similarly, such a mini-language lacks the great amount of detail contained in languages like PL/I and FORTRAN. Therefore, since he need not learn the various variable types, structures, and constructs of a full-blown language, the neophyte programmer can begin writing code to solve problems very soon after his introduction to the mini-language. It is this small size and simplicity which we have striven to design into the ROBOCAR language to allow users to concentrate on the solving of their problems rather than bogging down in the intricasies of the programming language they were using.

3. KEY CONCEPTS

To achieve our goal of educating the newcomer to computer science, we decided upon several concepts which we felt would comprise a reasonable background when mastered. These key items are listed and briefly discussed in this section and then detailed more closely in the next one. In that section, they are described in the context in which they are presented to the student.

Our overriding educational strategy was to present a sequence of related problems of slowly increasing complexity to the student. The sequence, allowing the student to proceed at his own pace and even to skip portions of it, builds upon itself as it proceeds to its conclusion. This approach can be seen at a macro level in the material in the four lessons and at a micro level within each lesson as the subjects are presented and developed.

The other concepts fall into three separate categories

problem-solving techniques,
language fundamentals,
and applications.

As with the size and simplicity characteristics of a mini-language, the three areas above are not really independent and separate. They provide, in this order, a way of looking at and organizing a method of attack for solving a programming problem; the means to construct a solution; and the

motivation to solve the problem in the first place. Although they will be presented in their developemental sequence in the next section, here the concepts will be grouped under the three categories above.

3.1 Problem Solving Techniques

First, let's consider problem solving techniques. The main concept here is the "debugging" approach to programming. Sample programs are developed and students are urged to consider the problem and try something which appears to be a reasonable solution, no matter how simple a start it is toward solving the problem. The next step (or steps depending on how quickly the program converges to a solution for the problem) is to refine and modify the procedure so that it moves closer to the desired result. As we will see in the next section, the lesson ROBOCAR, main instructional lesson in the sequence, is designed to facilitate this approach, Papert (1974), to computer problem solving.

Another useful technique is combining previously written programs to solve some new problem. After a user has mastered some of the language fundamentals discussed later in this section, he or she can use and is encouraged to combine old programs to solve new tasks.

Along the same lines, students are induced to think of pieces of previous programs as new instructions. By

considering old segments of code as functional entities, the programmer benefits in the following ways. He needn't rethink the correctness of the group of instructions he is using to perform a certain task. He must only take care to interface it properly with the rest of his current program. Consequently, the student gains in speed and productivity, and in the ability to cope with more challenging and rewarding problems.

3.2 Language Fundamentals

Language fundamentals is the second major category. The specific topics covered in this area range from primitive instructions to the construction of loops implemented using a stack. They are listed in the order they are presented in the instructional series of lessons.

Each computer and computer language will have a set of primitive or "built-in" instructions which form the basis for all programming. The ROBOCAR language is basically an assembly language for a computer which is termed the CAB; and thus preserves the one-to-oneness of primitive instructions to computer operations.

Besides learning the basic instructions through solving problems, the student learns that a computer program is merely a sequence of the instructions "understood" by the computer. Then, the new programmer encounters the concept

of a "straight-line" program built of primitive instructions with the flow of control proceeding one by one through the instructions from first to last.

Conditional and unconditional transfers of control open possibilities for solving new, more difficult and more interesting problems. The transfer instructions allow the student to program the computer to "remember" what has happened by branching to different portions of the program. By this time, the programmer is beginning to gather the tools necessary to make the computer perform the functions which make it the powerful tool that it is.

We move finally to the loop, which concludes the sequence on language fundamentals. Two methods of implementing loops are discussed and a few techniques and principles applying to both types appear. Initially, the user constructs loops which are continued on the basis of the truth or falsity of come condition.

An extreme case of this type of loop in the waiting loop, which has applications in operating systems as well as in other areas. The form used in the ROBOCAR series is

(THISLABEL) if (CONDITION) goto (THISLABEL).

The computer waits until the condition above is no longer true before it gives control to the next statement.

Following the introduction of a stack to the CAB computer, the concept of a loop executed a number of times dependent upon the initial value of a counter can be

presented. This implementation prepares the student for the "do loops" that he will encounter in the various higher level languages he employs in the future. Using the stack allows us to avoid the use of variables.

Commonly applicable to either type of loop mentioned before are the topics: nesting of loops and testing for exitting the loop. Nesting of simple loops necessitates no special insights except for the fact that it greatly increases the range of problems solvable by computer. However, implementing nested loops dependent on counters requires a thorough working knowledge of the operation of a stack: pushing, popping, and decrementing the top element, etc. Thus, the user gains another insight and practical tool in the world of computer science.

Finally, on the subject of looping, we emphasize the philosophy that it is better to test for an extreme condition before performing an operation that might exceed it. In other words, "Test first, then proceed." This maxim isn't claimed to apply to all situations, but should hold the programmer in good stead if he keeps it in mind.

3.3 Applications

The last category of key concepts is that programming skills and techniques do not exist in a vacuum. There must and do exist applications which present problems

solvable by computer. The first two lessons emphasize programming to solve problems which although interesting do not directly apply to "real-world" situations. The second two lessons implement the well-known "back-track" algorithm in the ROBOCAR language to introduce the student to another powerful programming technique. This demonstrates to him that extremely useful solutions can be programmed on a computer, even one as simple as the CAB equipped with the ROBOCAR language.

4. IMPLEMENTATION

Six lessons (four educational and two utility) implement the key concepts outlined in the previous section. In the preferred order of presentation, the four educational lessons are

ROBOCAR-----Introduction to programming in the ROBOCAR LANGUAGE,

ROBOSTACK---Introduction to counting loops, nested loops and applications via a stack,

MAZESEARCH--Background on the "back-track" algorithm, including three problem applications, and

ROBOBACK----Developement of the "back-track" algorithm in the ROBOCAR environment.

The utility programs are

ROBOEDIT----Interactive text-editor-compiler which allows students to enter programs in the ROBOCAR language, and

ROBOINT-----Interpreter which takes the program entered via ROBOEDIT and executes it, showing the results on the PLATO terminal screen thru the motion of the robot cab in a simulated city.

The diagram (Figure 4.1) on the next page summarizes the access from one program to another in the ROBOCAR series of lessons.

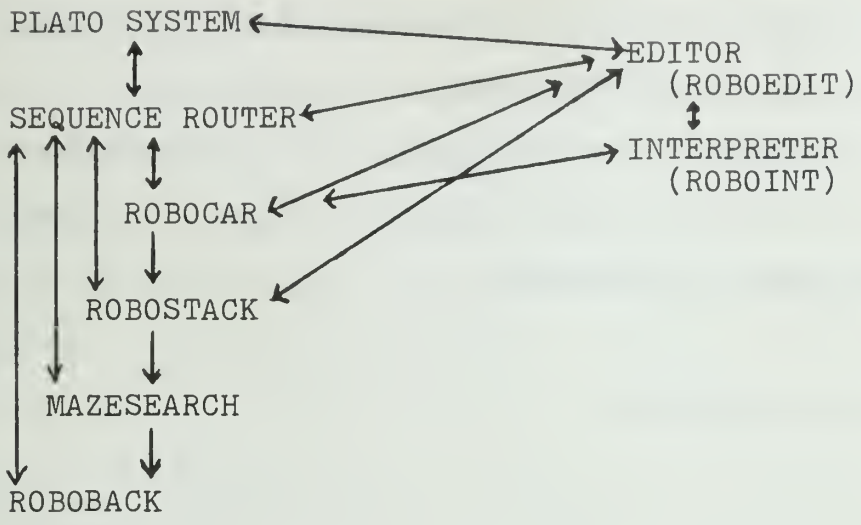


Figure 4.1 Access to Programs

As implied by Figure 4.1, the structure of the sequence is planned so that it carries out our educational goal of presenting a slowly increasing complexity on the program level. Also, by allowing the student to go back and forth between the editor and the interpreter, refining the program until the desired goal is reached, the simplistic trial and error method of attacking programming is fostered. Finally, a student can enter a program and execute it directly from the PLATO system so that he can forgo the complications of entering any of the other lessons in the sequence.

The remainder of this section will consist of a discussion outlining how each of the key concepts of SECTION 3 has been implemented.

Since it comprised the bulk of my programming time and is the vehicle for most of the concepts in the first two categories, the description of lesson ROBOCAR will go into more detail than those which follow on ROBOSTACK, MAZESEARCH, and ROBOBACK.

4.1 Lesson ROBOCAR

Alluded to but never fully explained, the ROBOCAR setup will be described first. The heart of our educational endeavor is the robot cab which acts as a computer, accepting lists of commands in the ROBOCAR language and exhibiting the results of these instructions by its movements around the city displayed on the screen. This approach was much influenced by the LOGO Project, Papert (1970), at the Massachusetts Institute of Technology. A list of the ROBOCAR language instructions and definition of their function is included in Appendix A.

When a ROBOCAR program is being executed, the program itself is displayed along with a pointer which points to each instruction as it is executed. Although the pointer moves very quickly from instruction to instruction, the student can easily tell if the program is in an infinite loop or if it is branching to the correct section of code at the appropriate time. For a more detailed view of the cab actions and which commands are prompting them, the user

has the option to execute his program in the step mode. In this mode, execution of the program follows its normal course, but execution of each succeeding instruction must be preceded by pressing a specified key.

These debugging aids are incorporated into the design to encourage the type of approach to computer problem solving mentioned earlier. The facilties just described, along with the link to the editor where the student can modify his current program, should give the student a large amount of exposure to absorb our key concepts.

PLATO screen images in the following three figures show the facilities and options available to the programmer as he

1) enters the editor to write a new program or modify his current one-----Figure 4.2,
2) enters the interpreter to see his program executed-----Figure 4.2,
3) views the debugging options available to him as his program runs-----Figure 4.3, and
4) watches his program being executed-----Figure 4.4.

After an introduction not unlike the one just finished, the remainder of the lesson ROBOCAR is organized into three distinct segments. They are a set of nine sample programs, a series of five programming problems, and a section called programming free-play where the student may go to the editor and interpreter to work on any programs he desires. Of these

three only the first two will be discussed, the third being self-explanatory in its function.

4.1.1 Sample Programs

In an attempt to allow users as much flexibility as possible, there is an index of sample programs from which one can branch to any one of the nine samples and to which one can go from any sample program explanation. Although the student can, in this manner, accelerate his pace through the lesson, the sample programs are ordered so that they build upon one another. By proceeding through them in the intended sequence, the student is introduced to each of the fundamental concepts, many of them being reinforced in subsequent explanations. Each explanation is structured in the same manner, again in a way giving the pupil the responsibility for and the capability to determine the course of action best suited to his or her needs as he understands them.

Figures 4.5 and 4.6 which follow will serve as a listing of the sample programs and options on accessing them and as a sample of how each program explanation appears to the student. After viewing them, then let us begin our discussion of the sample programs.

```
Begin editing.                          HELP available

>   a  fwd
       goto  a
```

Student's View Working on a Program

```
press next to execute program
press help for execution options
press back to leave

while your program is running you
may press back to return here.
```

Entering Interpreter to Execute a Program

Figure 4.2 Student Views and Options in Editor and Interpreter

Press the corresponding number of an option to select.

1) Execute in Single Step Mode
2) Display Entire Stack
3) Display Only Top Element of Stack
4) Display Instruction Counter
5) Display Stack Pointer
6) Display Cursor

press -data- to execute your program.
-back- to return to editor.

To reset an option, just press the key again.

Figure 4.3 Debugging Options During Execution

```
Your program:
a left
b  if  city_limits  goto  c
   fwd
   goto  a
c  right
   goto  b
```

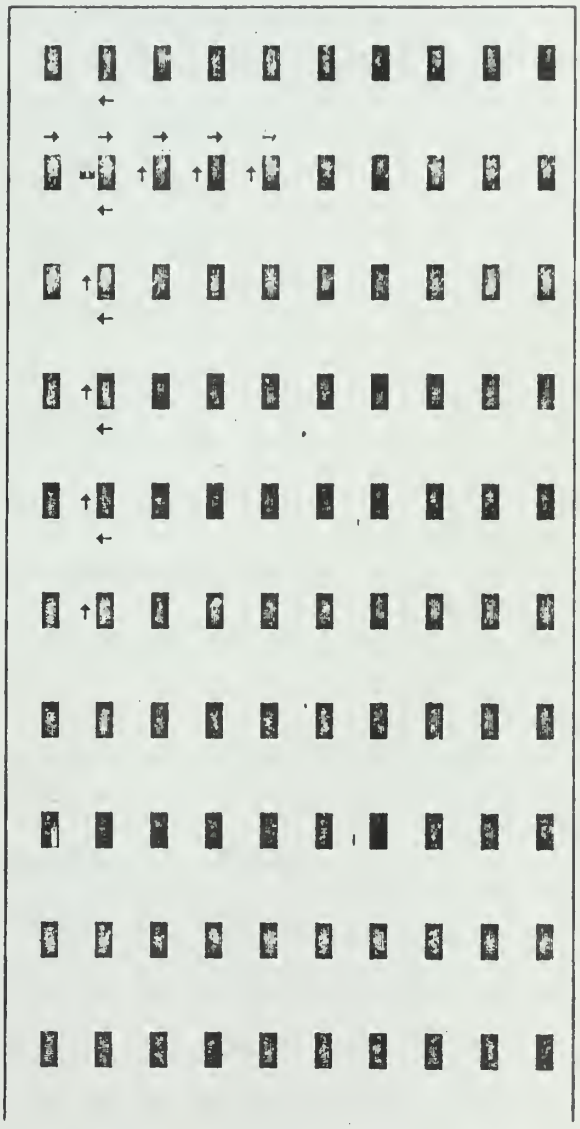


Figure 4.4 ROBOCAR Program Executing

2) Sample Programs Explained

The following programs, identified by their function, are explained in this section:

1) move cab forward 3 blocks
2) cab to city limits---1
3) cab to city limits---2
4) to first red light and stop
5) through all intersections, stop for red
6) to city limits, observing red lights
7) stop at red, turn right
8) as above, but watch for city limits
9) follow city limit all around city

Press the number of the program you wish to develope >

-NEXT- to proceed in sequence,
-BACK- to return to the introduction,
-LAB- for the programming problems,
-DATA- to return to the lesson index,
-STOP- to leave the lesson.

Figure 4.5 List of Sample Programs Explained

2) Sample Programs Explained

9) follow city limit all around city

Our problem here is to move the cab all around the city limits indefinately, assuming the cab starts with the city limits directly on its left.

Below is the program to do that:

```
a  left
b  if city_limits goto  c
   fwd
   goto  a
c  right
   goto  b
```

Press -NEXT- to see the program explained a line at a time

-LAB- to have the program executed

-BACK- to go back to the index of sample programs

Figure 4.6 Example of Sample Program

4.1.1.1 Sample Program 1

MOVE CAB FORWARD 3 BLOCKS

For the experienced programmer this program is trivial. To move the cab forward three blocks requires

```
fwd
fwd
fwd
stop*
```

Nevertheless, even here the new programmer learns several things. He is introduced to two of the ROBOCAR primitive instructions and is prepared for more upon having the concept of a "built-in" command explained to him. Also, he finds that a program consists of a sequence of valid instructions and that this program is a special case called a "straight-line" program. Now satisfied to himself that this program was trivial, he speeds right along the Program 2.

4.1.1.2 Sample Program 2

CAB TO CITY LIMITS--1

Immediately, the student is into a more difficult assignment. He must make the cab go forward from wherever it is to the city limits and stop there. The solution is

```
a  fwd
   if city_limits goto  b
   goto  a
b  stop
```

*see Appendix A if any commands are unfamiliar to you.

As we see, the student is quickly led to the use of conditional and unconditional transfers and to the idea of a loop.

The developement goes something like this, "Since we have to move the cab forward to the city limits, let's tell it to go forward one block. We then need to know if the cab is at the city limits." Of course, at this point, the conditional statement is brought in as the solution to the immediate problem. The assumption that the limits have been reached soon leads to the last statement in the program. Naturally, the use and rationale for labels is introduced here. Now the student must confront the possibility that one move forward didn't put the cab at the city limits.

Let's pick up the conversation again, "What if the cab is not at the city limits after one move forward? (pause) We must move the cab forward again and check for the city limits, right?" An X is then marked in the next line down from the conditional. The dialogue continues by reasoning out what must be done at point X in the program. Pursuing a solution at which a mature programmer would snicker, the lesson suggests putting a copy of the first two statements in the program at point X. The student may be happy with that until the question, "What if the city limits was three, four or more blocks away?" is raised. The unconditional goto makes its appearance at the appropriate spot after the

student (hopefully) agrees that going back to the fwd statement will solve the problem for an arbitrarily long distance from the city limits.

Finally, the loop in Program 2 is identified and defined as a loop. The possibility of a special loop called an infinite loop is brought up and the student, possibly nervously, moves on to Program 3.

4.1.1.3 Sample Program 3

CAB TO CITY LIMITS--2

To his relief, he sees here an old friend. However, he soon discovers that there is a situation which causes the cab to fail to stop when it is at the city limits. The presentation shows him the program from the previous problem, telling him that there exists a situation as described above. He is asked, then, "In what starting position will the cab not reach the city limits and stop?" Following a pause, the program acknowledges that the student probably saw that the program would fail if the cab was at the city limits facing it to begin with. Consequently, the programmer leaves Program 3 with another look at the now familiar loop structure, and with the advice of testing first, then going ahead.

4.1.1.4 Sample Program 4

TO FIRST RED LIGHT AND STOP

Program 4 is analogous to number three except a new conditional, the if red conditional, is introduced which replaces the if city_limits instruction in order to stop the cab at the first red light. By now, how and when to employ loops with conditional exits should be jelling in the programmer's mind.

4.1.1.5 Sample Program 5

THROUGH ALL INTERSECTIONS, STOP FOR RED

Waiting and nested loops make their appearance in Program 5 although no new instructions are introduced. The three statement program below serves to illuminate these concepts

```
a  if red goto  a
   fwd
   goto  a
```

Since the goal is to go forward through all intersections, waiting for any red lights, the programmer is asked how to start building the program. If he remembers the principle, "check first, then proceed", he will say, "check to see if the cab faces a red light at its current intersection." If there is a red light the cab must wait for the light to turn green. He no doubt agrees that the "if red" question must be asked again, prompting the developement of the

waiting loop. The explanation cautions him that the waiting loop must be able to sample the condition upon which it is waiting at a much faster rate than the condition changes. He finds, further, that there are two loops here, one nested within the scope of the other.

4.1.1.6 Sample Program 6

TO CITY LIMITS, OBSERVING RED LIGHTS

In Program 6, the student sees the first real synthesis of two programs. As he noticed when he executed the last program, the cab stopped after it had penetrated the city limits, the message, "Sorry, cab out of city limits" appearing on the screen. "This time," goes the discussion, "We would like to write a program which is a combination of the goals of Programs 3 and 5." The programs are listed

```
a  if city_limits goto  b
   fwd
   goto  a
b  stop
```

PROGRAM 3

```
c  if red goto  c
   fwd
   goto  c
```

PROGRAM 5

and the user is asked how the programs can be combined to allow the cab to go to the city limits and stop, while waiting for all red lights it encounters. With a little thought and prompting, the student sees that all Program 3

lacks is for the cab to wait until the light is no longer red before it ventures forward.

Since the first two instructions of Program 5, the red light waiting loop and the fwd, have allowed the cab to pass legally through an intersection, they can be used to replace the fwd in Program 3, allowing the conclusion that this pair of instructions can be thought of as a single instruction which sends the cab through any intersection with proper observance of the traffic lights. The programmer will no longer have to rethink the correctness of this code. Therefore, because Program 3 stopped the cab at the city limits, the two programs have been combined to achieve the stated goal.

4.1.1.7 Sample Program 7

STOP AT RED, TURN RIGHT

Because the traffic lights change more or less randomly, Program 7 turns out to be an example of a random walk implemented in the ROBOCAR language. A section of code, now familiar, is written which moves the cab forward until it comes to a red light. Then control is transferred to another part of the program. Obviously, at that point, the cab is to turn right and then start the whole process again. The programmer soon realizes that different positions of control in the program can be used to represent different

situations of the cab.

4.1.1.8 Sample Program 8

AS ABOVE, BUT WATCH FOR CITY LIMITS

Program 8 is another synthesis problem, building on problem 7. A new test to check for the city limits must be added before the cab can go forward; otherwise, the cab could violate the out of city limit rule as it might have in Program 7. In this manner, another situation must be remembered by branching. When at the city limits the cab must turn right until it no longer faces that boundary.

4.1.1.9 Sample Program 9

FOLLOW CITY LIMIT ALL AROUND CITY

Program 9 has a twofold purpose. First, it is evidence that a reasonably complex task can be achieved using the few instructions that the student has employed in the earlier programs. Second, it is an attempt to get the programmer to confront and verify the correctness of the algorithm and the code which he utilizes to solve a particular problem. Once again, the problem is to move the cab around the city limits of any arbitrarily shaped city with the city limits always one block to the left, Papert and Soloman (1972). For simplicity, the cab starts

with the city limits directly to the left.

The strategy presented to the student is, "Turn left and go forward if you can; if not, keep turning right until you can go forward." Only six statements comprise the program which is subsequently developed line by line in a straight-forward manner. Incidentally, Program 9 appears in Figure 4.6.

To demonstrate the correctness of the program, a simulation showing the three different situations in which the cab may start the program is performed. Much as he sees when programs are executed in the interpreter, the student watches as the cab works its way out of each situation. An arrow points from one instruction to the next one performed as the cab proceeds. Since the pointer returns to the beginning statement after each situation and at any time the three situations are all that can immediately face the cab, the algorithm is, we hope, proven to the student.

We have now completed the explanation of the sample programs explained portion of lesson ROBOCAR. The other major segment remaining is the programming problems section.

4.1.2 Programming Problems

The introduction that the student receives when he accesses this portion of the lesson will serve as well as

any other way to describe the intent and contents of the programming problems section. It follows in Figures 4.7 and 4.8.

Next, the list of programming problems from the lesson is shown in Figure 4.9. This figure is an image of the PLATO screen when the student is ready to pick a programming problem on which to work.

The presentation of one of the programming problems is the topic of the next figure, Figure 4.10. In this picture you can see the options available to the student as he attempts to solve the particular problem he is on. Figure 4.10 shows the action after the student has gone to the editor and attempted a solution to the problem and then come back to the programming problem to seek help in the form of the suggested solution. Referring back to Figures 4.2 through 4.4 will remind you of the options and opportunities the programmer has when he accesses the editor and the interpreter.

At this point, then, the description of lesson ROBOCAR draws to a close, leaving only abstracts of the lessons ROBOSTACK, MAZESEARCH, and ROBOBACK remaining in this section.

3) Programming Problems

0) Introduction

A useful approach in solving programming problems is the "top-down" method.

By top-down, we mean that one developes a general plan of attack represented by a sequence of instructions that can reasonably be expected to perform the task desired.

This code is executed and the performance of the program is compared to the goal behavior desired.

If the result is not exactly as we want, then we must refine the program in finer detail to more closely achieve the proper action.

Press -HELP- for an example of top-down method,
-NEXT- for more introduction,
-DATA- for the first programming problem,
-BACK- for the index of problems.

Figure 4.7 Introduction to Programming Problems

3) Programming Problems

0) Introduction

As far as the type of problems presented, they vary from minor modifications of the sample programs, explained in the last section, to problems completely different in content.

For the former types, you may find it useful to refer back to an appropriate program in the last section.

For your use in writing these programs, you may have the current version of your program printed on the screen.

Furthermore, if you have completed the program, or are just plain stymied, you can request a listing of a solution to the problem.

Press -NEXT- for the first programming problem,
-BACK- for the index of problems.

Figure 4.8 Introduction to Programming Problems (continued)

3) Programming Problems

PROGRAMMING PROBLEMS INDEX

0) Introduction--approach to problems
types of problems
aids in solving problems

1) Problem 1--forward three blocks and back to start

2) Problem 2--straight to city limit and back to other side

3) Problem 3--stop at red, turn right; but only on green

4) Problem 4--at every red light, turn left, go fwd, turn left

5) Problem 5--go to city limits then turn left and follow around city

Press the number of the section you wish to try >

-or-

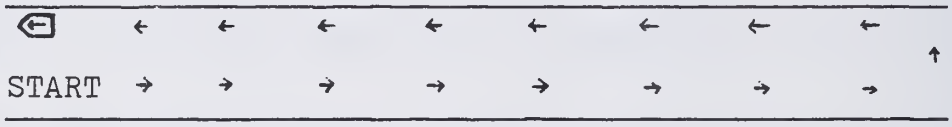
-NEXT for the introduction,
-BACK- for the sample programs,
-LAB- for the programming freeplay,
-DATA- for the lesson table of contents,
-STOP- to leave the lesson.

Figure 4.9 Index of Programming Problems

3) Programming Problems

1) Problem 1

Your task here is to move the cab forward three blocks; turn it around; and bring it back even with the starting point heading opposite the original direction.



SUGGESTED SOLUTION:

```
fwd
fwd
fwd
left
left
fwd
fwd
fwd
stop
```

YOUR SOLUTION:

```
a  fwd
   fwd
   fwd
   left
   goto  a
```

Press -LAB- to enter your program solution,
-NEXT- to see your program as it is now,
-DATA- to work the next problem,
-BACK- to see the index of problems, or
-HELP- to see the suggested solution.

Figure 4.10 Example Problem Presentation

4.2 Lesson ROBOSTACK

As mentioned earlier in this section and in Section 3, lesson ROBOSTACK introduces the cab as a "stack machine" opening the possibility of implementing loops with counters. The program starts with some background on these loops and then proceeds to show how they can be implemented using a stack. Of course, the necessary commands have been incorporated into the ROBOCAR language and are introduced to the student at this time.

4.2.1 Loop Developement

A simple loop executed one hundred times is explained and implemented by the following code:

	INSTRUCTION	COMMENT
	push 100	create a new top element of the stack equal to 100
a	dec	subtract 1 from top element
	if zero goto b	if loop has gone 100 times exit to statement labeled b
	goto a	if not, go again
b	pop	remove top element of stack to access next element down

After this implementation is discussed, the point is made that another loop can be nested inside the first one.

The student then has the opportunity to see an explanation of two applications and to do some problems involving nested loops.

4.2.2 Applications

Four nested loops make up the hours, minutes, seconds and cpu cycles per second which implement a simulated clock. The clock simulation is the first application discussed, the second of which is making the cab do an inward clockwise spiral.

For the spiral, the stack is initially set with the number of sides or "arms" on the spiral which also turns out to be the length of the first "arm" forming the outer loop. The inner loop pushes the current value of the top element on the stack to the top. This action actually duplicates the top element as the new top element, thus it is made a command in the ROBOCAR language and called the dup instruction. After moving the cab forward the prescribed number of blocks, the inner loop passes control to the outer loop which turns the cab right, pops the stack, decrements the old top element and starts a new arm.

4.2.3 Problems

The student is then invited to do a square, after which

he is probably ready to try counterclockwise outward spirals and other creative endeavors, Papert and Soloman (1972). The final two lessons, as previously stated, introduce and develope the "back-track" algorithm in several different contexts.

4.3 Lesson MAZESEARCH

The lesson MAZESEARCH developes the "back-track" algorithm in a visual manner by showing how the ancient Greek hero Theseus might have used it to search the Labyrinth exhaustively in hopes of slaying the mythological Minotaur.

4.3.1 Theseus and the Minotaur

Theseus unwinds a string along the path he takes and uses his sword to mark the entrances to the pathways he has traversed. These two items, of course, form the signal components of the back-track algorithm: the ability to retrace one's steps and to mark in some way those paths that are taken. It is worth remembering that these two functions have had to have been incorporated into the ROBOCAR environment for it to support the implementation of the algorithm in lesson ROBOBACK, which is discuseed later.

Given the two functions above, Theseus devises a

flowchart for the algorithm and at the end intimates that it looks like it could be used for a lot of different problems. Of course it can, and after a clever animation of Theseus searching the maze and slaying the Minotaur, two new applications are presented.

The student learns that the chess queens problem and a coin changing problem are both suited to the exhaustive search of a system of nodes and pathways which the back-track algorithm provides.

4.3.2 Chess Queens Problem

The chess queen problem is that of placing n queens on an n x n chessboard ($n \geq 4$) so that no queen can capture anyother queen in one move if all others retain their original placements. A flowchart, based on the back-track algorithm but modified into the context of the queens problem, is developed and a segment where the student may choose the size of board and placement of queens and have the search done for him is provided.

4.3.3 Coin Changing Problem

The developement for the coin changing problem, where all possible combinations of one dollar's worth of change must be listed, is analogous to that of the queens problem. Therefore no more discussion will come for it.

4.3.4 Maze Designing

The final portion of MAZESEARCH lets the student design mazes of his own choosing (subject to a few restrictions) through which a mouse attempts to find his way. One can, at appropriate times, stop the mouse, redesign the latest maze, or draw a whole new one.

Following the developement and applications of the back-track algorithm in lesson MAZESEARCH, lesson ROBO-BACK implements the algorithm in the ROBOCAR language environment.

4.4 Lesson ROBOBACK

Again, in this lesson, the back-track idea is used to build a flowchart, and eventually a program, to solve an exhaustive search problem. The statement of the problem comes directly from the lesson:

> Last night in PLATO city there was a totally unpredictable snowstorm. Twelve inches of snow fell, leaving the inhabitants immobilized and cut off from help.
> Fortunately, there is a snowplow in the city that can be used to clear the streets and free the inhabitants. You are the operator. So it's up to you to guide the plow through all the streets of PLATO city, using commands in the ROBOCAR language.
>
> Before you start your heroic mission, however, you have to keep in mind these two things:
>
> 1. The plow can only clear one side of the street at a time, so you have to plow the same street twice (once from each way).

2. Due to the gas shortage, there is only enough gas for your plow to go over each street twice. This means that you can't go over a street that has already been plowed twice.

As soon as he has read the statement of the problem, the student is warned that he must have a thorough understanding of both the back-track algorithm and the ROBOCAR language. He is encouraged to go back and review any lessons he needs to in order to bolster his background in these areas. When he is ready to proceed, the back-track algorithm is applied to the snowplow problem in great detail and patience. At the end of the lesson, the program performs an attractive simulation of the snowplow plowing the city using the modified algorithm.

This concludes the description of the ROBOCAR sequence implementation. We sincerely hope that the key concepts of section 3 will be and have been adequately conveyed to the students past, present and future. And we further hope that the spin-offs such as flowcharting and computer terminology will be helpful.

Above all, it is our desire that these students will be better prepared for whatever involvement the world of computers and computer science will have in their future, having a feeling that they better understand and can control the powerful computer which, in the end, merely follows to the letter the instructions with which it is programmed.

APPENDIX A

ROBOCAR LANGUAGE SUMMARY

INSTRUCTION		MEANING
CAB MOVEMENT:		
1)	fwd	: moves cab one block in the direction the cab is heading. If the cab goes past the city limits the program halts in error.
2)	left	: rotates the cab 90° counterclockwise. The cab makes no forward movement.
3)	right	: rotates the cab 90° clockwise. The cab makes no forward movement.
4)	east	: rotates the cab until it is heading east (right side of screen) with no forward movement.
5)	west	: same as east except cab heads west (left side of screen).
6)	north	: same as east except cab heads north (top of screen).
7)	south	: same as east except cab heads south (bottom of screen).
STACK OPERATIONS:		
8)	dup	: creates new top element of stack which is a copy of the previous top.
9)	dec	: decrement the top of the stack by one.
10)	pop	: delete top element of stack (causes error if stack is empty).

APPENDIX A

INSTRUCTION	MEANING
STACK OPERATIONS:	
11) push x	: adds new top element of stack whose value must be between -131071 and 131072.
12) pushih	: makes the new top element of the stack the opposite of the direction the cab is heading. This is used in the back-track method.
13) execute_top	: makes the next instruction executed be the heading which is the top element of the stack. The stack is not popped and an empty stack will halt the program in error.
CONTROL:	
14) goto x	: control is transferred to the statement labeled x.
15) stop	: ends execution of program.
CONDITIONAL:	
16) if red goto x	: if the intersection ahead has a red light in the direction the cab is heading, control passes to the statement x; otherwise, the next command below is executed.
17) if city_limits goto x if limits goto x	: if there is no street in the direction the cab is heading transfer control to statement x; else next statement below.

APPENDIX A

INSTRUCTION		MEANING
CONDITIONAL:		
18)	if unplowed goto x if unmarked goto x	: if the intersection ahead has not been traversed from this direction, go to x; else, to next command.
19)	if stack_empty goto x if empty goto x	: if the stack is empty go to x; else to the next statement.
20)	if zero goto x	: if the top element of the stack equals zero, go to x; else go to next command.

APPENDIX B

ROBOCAR TEXT-EDITOR COMMANDS

(help page)

TEXT-EDITOR COMMANDS

	KEY TO ACTIVATE
ENTER LINE (cursor at blank line):	
1) Start entering new line	none
LINE-EDITING (middle of line):	
2) Start over on line	EDIT
CURSOR MOVEMENT:	
3) UP	BACK
4) DOWN	DATA
LINE-EDITING (beginning of line):	
5) REPLACEMENT	EDIT
6) DELETION	ERASE
7) INSERTION (below first line)	LAB
8) INSERTION (above first line)	shift-LAB
END OF PROGRAM-ENTRY:	
9) Stop editing	STOP

Press BACK to edit again, the number of the command for further explanation:

APPENDIX C

ROBOCAR LANGUAGE COMMANDS

(help page)

ROBOCAR LANGUAGE SUMMARY

INSTRUCTION — MEANING

CAB MOVEMENT:
1) fwd
2) left
3) right
4) east
5) west
6) north
7) south

west: rotates the cab until it is heading west with no cab movement
Press NEXT

STACK OPERATIONS:
8) dup
9) dec
10) pop
11) push x
12) pushih
13) execute_top

CONTROL:
14) goto x
15) stop

CONDITIONAL:
16) if red goto x
17) if (limits or city_limits) goto x
18) if (unmarked or unplowed) goto x
19) if (empty or stack_empty) goto x
20) if zero goto x

Press BACK to leave or the number of the instruction

to be explained: 5 ok

APPENDIX D

LESSON IDENTIFICATION AND SIZE

LESSON	SINGLE OR DOUBLE LESSON SPACE	SPACE IN WORDS
ROBOEDIT	SINGLE	3701
ROBOINT	SINGLE	3392
ROBOCAR	DOUBLE	8076
ROBOSTACK	SINGLE	2854
MAZESEARCH	DOUBLE	8103
ROBOBACK	SINGLE	3406

REFERENCES

Nievergelt, J. (1975), "Interactive Systems for Education--The New Look of CAI", IFIP World Conference on Computer Education.

Papert, S. (1970), "Teaching Children Thinking", IFIP World Conference on Computer Education, 73-78.

Papert, S. (1974), Seminar on Education, University of Illinois, Urbana, January 14-16, 1974.

Papert, S. and Soloman, C., "20 Things To Do With a Computer", Educational Technology, 12, 4 (April, 1972), 9-18.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-75-741	2.	3. Recipient's Accession No.
4. Title and Subtitle ROBOCAR: EDUCATING THE LAYMAN IN COMPUTER SCIENCE			5. Report Date July, 1975 6.
7. Author(s) James W. Renshaw			8. Performing Organization Rept. No.
9. Performing Organization Name and Address Department of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois			10. Project/Task/Work Unit No. 11. Contract/Grant No. NSF EC-41511
12. Sponsoring Organization Name and Address National Science Foundation Washington, D.C.			13. Type of Report & Period Covered 14.

15. Supplementary Notes

16. Abstracts

ROBOCAR is a series of lessons on the PLATO IV System designed to teach computer programming in an environment which allows fundamental concepts of programming to be illustrated in an intuitive setting and promotes interactive practice through the use of animated graphics controlled by a text-editor and interpreter.

17. Key Words and Document Analysis. 17a. Descriptors

interpreter
text editor

17b. Identifiers/Open-Ended Terms

computer assisted instruction
man machine systems

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages
	20. Security Class (This Page) UNCLASSIFIED	22. Price